BSTZ No. 042390.P10802
Express Mail No. EL802873351US

UNITED STATES PATENT APPLICATION

FOR

PRUNING LOCAL GRAPHS IN AN INTER-PROCEDURAL ANALYSIS SOLVER

Inventor:
Arch D. Robison

Prepared by:

Blakely, Sokoloff, Taylor & Zafman LLP
12400 Wilshire Boulevard, Suite 700
Los Angeles, California 90025
(714) 557-3800

PRUNING LOCAL GRAPHS IN AN INTER-PROCEDURAL ANALYSIS SOLVER

BACKGROUND

FIELD OF THE INVENTION

[0001] This invention relates to compiler technology. In particular, the invention relates to inter-procedural analysis.

BACKGROUND OF THE INVENTION

[0002] A compiler translates a source program to one or more object files. The source program may contain one or more translation units. A translation unit is a subroutine, a function, or any other separately compilable software entity. A compiler typically includes a front end and a back end. The front end typically performs lexical and syntactic analysis, creates symbol table, and generates intermediate code. The back end typically performs code optimization and generates the target object files. Inter-procedural analysis (IPA) is a phase in a compilation process to analyze the entire program and collect global information related to the translations units. The collected global information is then passed to the optimizer for global optimization.

[0003] Distributed IPA processes files on disk. When the file size is large, the disk access time may be excessive, resulting in slow compilation. When separately compilable software entities are represented by some data structures, the amount of storage may be large if the size or the number of the software entities is large. For distributed IPA, there may be several locally generated data structures to be written to and read from mass storage devices. Since access time

for mass storage devices is slow, the large amount of information exchange via the mass storage may lead to inefficient usage.

[0004] Therefore, there is a need for a technique to reduce disk storage requirements and improve access time during compilation.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The features and advantages of the present invention will become apparent from the following detailed description of the present invention in which:

[0006] Figure 1 is a diagram illustrating a system in which one embodiment of the invention can be practiced.

[0007] Figure 2 is a flow chart illustrating a process to reduce storage in inter-procedural analysis solver according to one embodiment of the invention.

[0008] Figure 3 is a flow chart illustrating a process to prune local graphs according to one embodiment of the invention.

[0009] Figure 4A is a flowchart illustrating a process to shrink the local graphs according to one embodiment of the invention.

[0010] Figure 4B is a diagram illustrating an edge removal in shrinking the local graphs according to one embodiment of the invention.

[0011] Figure 4C is a diagram illustrating a subgraph transformation in shrinking the local graphs according to one embodiment of the invention.

[0012] Figure 5 is a pseudo code illustrating a process to associate a use attribute shown in Figure 3 according to one embodiment of the invention.

[0013] Figure 6 is a pseudo code illustrating a process to associate an affect attribute shown in Figure 3 according to one embodiment of the invention.

[0014] Figure 7 is a diagram illustrating an example for the pre-solving shown in Figure 3 according to one embodiment of the invention.

[0015] Figure 8 is a diagram illustrating an example of a three-point value lattice according to one embodiment of the invention.

[0016] Figure 9 is a diagram illustrating an example in the C programming language of two translation units according to one embodiment of the invention.

[0017] Figure 10 is a diagram illustrating the local graphs for the two translation units shown in Figure 9 according to one embodiment of the invention.

[0018] Figure 11 is a diagram illustrating the transfer functions of the local graphs shown in Figure 10 according to one embodiment of the invention.

[0019] Figure 12 is a diagram illustrating pruning the local graphs shown in Figure 10 according to one embodiment of the invention.

[0020] Figure 13 is a diagram illustrating the transfer functions of the pruned local graphs shown in Figure 12 according to one embodiment of the invention.

[0021] Figure 14 is a diagram illustrating another example for pruning the local graphs according to one embodiment of the invention.

[0022] Figure 15 is a diagram illustrating a process to pre-solve a subgraph according to one embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0023] One embodiment of the invention is a pruning technique to reduce the size of the local graphs used in IPA solver for separately compilable software entities. The technique reduces the disk storage requirements and input/output overhead for files used for distributed IPA.

[0024] In the following description, for purposes of explanation, numerous details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that these specific details are not required in order to practice the present invention. In other instances, well-known electrical structures and circuits are shown in block diagram form in order not to obscure the present invention.

[0025] Figure 1 is a diagram illustrating a computer system 100 in which one embodiment of the invention can be practiced. The computer system 100 includes a processor 110, a host bus 120, a memory control hub (MCH) 130, a system memory 140, an input/output control hub (ICH) 150, a mass storage device 170, and input/output devices 180_1 to 180_K.

[0026] The processor 110 represents a central processing unit of any type of architecture, such as embedded processors, micro-controllers, digital signal processors, superscalar computers, vector processors, single instruction multiple data (SIMD) computers, complex instruction set computers (CISC), reduced instruction set computers (RISC), very long instruction word (VLIW), or hybrid

architecture. In one embodiment, the processor 110 is compatible with the Intel Architecture (IA) processor, such as the IA-32 and the IA-64. The host bus 120 provides interface signals to allow the processor 110 to communicate with other processors or devices, e.g., the MCH 130. The host bus 120 may support a uni-processor or multiprocessor configuration. The host bus 120 may be parallel, sequential, pipelined, asynchronous, synchronous, or any combination thereof.

[0027] The MCH 130 provides control and configuration of memory and input/output devices such as the system memory 140 and the ICH 150. The MCH 130 may be integrated into a chipset that integrates multiple functionalities such as the isolated execution mode, host-to-peripheral bus interface, memory control. For clarity, not all the peripheral buses are shown. It is contemplated that the system 100 may also include peripheral buses such as Peripheral Component Interconnect (PCI), accelerated graphics port (AGP), Industry Standard Architecture (ISA) bus, and Universal Serial Bus (USB), etc.

[0028] The system memory 140 stores system code and data. The system memory 140 is typically implemented with dynamic random access memory (DRAM) or static random access memory (SRAM). The system memory may include program code or code segments implementing one embodiment of the invention. The system memory 140 includes a local graph pruning module 142 and an inter-procedural analysis (IPA) solver 144. The system memory 140 may also include other programs or data, which are not shown depending on the various embodiments of the invention. The instruction code stored in the memory 140, when executed by the processor 110, causes the processor to perform the tasks or operations as described in the following.

[0029] The ICH 150 has a number of functionalities that are designed to support I/O functions. The ICH 150 may also be integrated into a chipset together or separate from the MCH 130 to perform I/O functions. The ICH 150 may include a number of interface and I/O functions such as PCI bus interface, processor interface, interrupt controller, direct memory access (DMA) controller, power management logic, timer, universal serial bus (USB) interface, mass storage interface, low pin count (LPC) interface, etc.

[0030] The mass storage device 170 stores archive information such as code, programs, files, data, applications, and operating systems. The mass storage device 170 may include compact disk (CD) ROM 172, floppy diskettes 174, and hard drive 176, and any other magnetic or optic storage devices. The mass storage device 170 provides a mechanism to read machine-readable media. The mass storage device 170 may also be used to store representations of the separately compilable software components such a local graphs as generated by the local graph pruning module 142 and the IPA solver 144.

[0031] The I/O devices 180_1 to 180_K may include any I/O devices to perform I/O functions. Examples of I/O devices 180_1 to 180_K include controller for input devices (e.g., keyboard, mouse, trackball, pointing device), media card (e.g., audio, video, graphics), network card, and any other peripheral controllers.

[0032] The present invention may be implemented by hardware, software, firmware, microcode, or any combination thereof. When implemented in software, firmware, or microcode, the elements of the present invention are the program code or code segments to perform the necessary tasks. A code segment may represent a procedure, a function, a subprogram, a program, a routine, a subroutine, a module, a software package, a class, or any combination of

instructions, data structures, or program statements. A code segment may be coupled to another code segment or a hardware circuit by passing and/ or receiving information, data, arguments, parameters, or memory contents. Information, arguments, parameters, data, etc. may be passed, forwarded, or transmitted via any suitable means including memory sharing, message passing, token passing, network transmission, etc. The program or code segments may be stored in a processor readable medium or transmitted by a computer data signal embodied in a carrier wave, or a signal modulated by a carrier, over a transmission medium. The "processor readable medium" may include any medium that can store or transfer information. Examples of the processor readable medium include an electronic circuit, a semiconductor memory device, a ROM, a flash memory, an erasable ROM (EROM), a floppy diskette, a compact disk CD-ROM, an optical disk, a hard disk, a fiber optic medium, a radio frequency (RF) link, etc. The computer data signal may include any signal that can propagate over a transmission medium such as electronic network channels, optical fibers, air, electromagnetic, RF links, etc. The code segments may be downloaded via computer networks such as the Internet, Intranet, etc.

[0033] It is noted that the invention may be described as a process, which is usually depicted as a flowchart, a flow diagram, a structure diagram, or a block diagram. Although a flowchart may describe the operations as a sequential process, many of the operations can be performed in parallel or concurrently. In addition, the order of the operations may be re-arranged. A process is terminated when its operations are completed. A process may correspond to a method, a function, a procedure, a subroutine, a subprogram, etc. When a process corresponds to a function, its termination corresponds to a return of the function to the calling function or the main function.

[0034] The local graph pruning module 142 and the IPA solver 144 are used to optimize the IPA process for separately compilable software entities. The approach is based on a lattice-theoretic framework for IPA of programs consisting of more than one separately compiled translation units. Local inter-procedural problems are constructed for each translation unit, reduced, and merged together into a global problem to be solved. Local solutions are derived from the global solution, and used to optimize each translation unit when it is recompiled. All problems and solutions are formulated over lattices in a way that allows the framework to automatically determine when files are either recompiled (for correctness) or optionally recompiled (for possible improvement). The approach is based on graph theory and discrete mathematics with concepts in partial ordering, lattice, and lattice attributes.

[0035] A partial order is a relation, signified by the symbol $\leq$ having the following properties:

Transitive: $x \leq y$ and $y \leq z$ implies $x \leq z$.

Reflexive: $x \leq x$ is always true

Anti-symmetric: $(x \leq y)$ implies that either $x = y$ or not $(y \leq x)$

[0036] For example, the relation "is a divisor of" is a partial order for positive integers; the relation "is less than or equal to" is a partial order for integers; the relation "is a subset of" is a partial order if each element is a set. The ordering is "partial" because not all pairs of elements can be compared. For example, 2 is not a divisor of 3, nor vice-versa. When dealing with a partial order, for any two elements x or y, one of the following four situation holds:

$x = y$ is true

$x \leq y$ is true but $y \leq x$ is false

$y \leq x$ is true but $x \leq y$ is false

Both $x \leq y$ and $y \leq x$ are false

In the last case, the values are "incomparable".

[0037] The solutions are a monotone function of the problems: for two problems p and p' with respective solutions s and s', then $p \leq p'$ implies $s \leq s'$.

[0038] Typically, partial orders are lattices. A lattice is a partial ordering closed under the operations of a least upper bound and a greatest upper bound. The "meet" of a set of elements is an element that is less than or equal to every element in the set. For example, let "$\leq$" denote "is a divisor of". Then given {12, 24, 30}, the meet is 6, because 6 is a divisor of each element in the set and there is no larger divisor. 3 is a lower bound divisor, but since it is a divisor of 6, it is not the greatest. The "join" is an element that is greater than or equal to every element in the set. For "is a divisor of", the "join" is simply the least common multiple. Closed means that the bounds exist in the set under discussion. For example, if the set were composite (non-prime) numbers only, then the "meet" of {12, 15}, which is 3, would not be in the set.

[0039] The "top" of a lattice is the element that is the join for the set of all elements. The "bottom" is the meet for the set of all elements. Thus, "top" and "bottom" are the identity elements for "meet" and "join", respectively. For example, infinity is the top of the divisor lattice, and 1 is the bottom of that lattice.

[0040] A function f that maps a lattice of values onto itself is monotone if $x \leq y$ implies $f(x) \leq f(y)$ for any two lattice elements x and y. The set of monotone functions over a lattice of values form a lattice of functions, where $f \leq g$ if and only of $f(x) \leq g(x)$ for all lattice values x.

[0041] The composition f o g of two functions is defined by the relation $(f \circ g)(x) = f(g(x))$ for all x in the domain of g.

[0042] The usual form for representing a problem and a solution is a directed graph. A directed graph is a set of vertices and a set of directed edges. Each edge connects its tail vertex to its head vertex. An edge from vertex u to vertex v is denoted $u \rightarrow v$. Each vertex of the graph has a lattice value, and each edge has a monotone lattice transfer function. Value of a vertex u is denoted as u.val or val(u). Transfer function of an edge e is denoted as e.func or func(e).

[0043] Figure 2 is a flow chart illustrating a process 200 to reduce storage in inter-procedural analysis solver according to one embodiment of the invention.

[0044] Upon START, the process 200 creates a local problem p_i for each translation unit i (Block 210). For N translation units, there are N local problems. The set of all possible problems form a partial order. The set of all possible solutions form a partial order. Next, the process 200 represents the local problems p_I's by local graphs (Block 220). Then, the process 200 prunes the local graphs (Block 230). The pruned local graphs provide reduced storage requirements and speed up input/output (I/O) transfer time when files are written and read from disk.

[0045] Next, the process 200 forms a global graph from the pruned local graphs (Block 240). The global graph represents a global problem. Then, the

process 200 solves the global problem using an inter-procedural analysis (IPA) solver. The process 200 is then terminated.

[0046] The IPA solver may include the following steps: (1) create a global problem P from the local problems such that $P \leq p_i$ for all i; (2) compute the solution S for the global problem P. The map from problems to solutions is monotone; (3) set each local solution s_i such that $s_i \leq S$ for all i; (4) use solution s_i to optimize translation unit i.

[0047] The local graphs are pruned or reduced and then merged together to form a global graph. The merging process in the IPA solver merges vertices with identical names. Anonymous vertices are never merged. Duplicate edges between the same pair of named vertices are merged into a single edge between the same vertices with a transfer function that is the lattice-meet of the transfer functions for the duplicates. A greatest fix-point global solution is then computed for the global graph, and some values for named vertices of the global are reported back to the local compilations. The greatest fix-point solution is a mapping of vertices to values, denoted SOL(v), where:

For all vertices, $SOL(v) \leq v.val$

For all edges e of the form $u \rightarrow v$: $SOL(v) \leq e.func(SOL(u))$

There is no other SOL'(v) such that $SOL(v) \leq SOL'(v)$

[0048] Condition (c) is not essential for correctness, but is merely preferred for sake of best optimization.

[0049] A local compilation of x.o sends a file x.opa to the IPA solver and the IPA solver sends back a file x.ipa. The visible part of the local problem is sent

to the IPA solver via a "problem" section in the .opa file. The visible part is that which might affect the global solution. The technique in the invention reduces the size of the local graphs, or prunes the graphs, without changing the global solutions for the named vertices. This results in reduced size and I/O overhead for the .opa file. The IPA solver reads all the local problems p_i's from the "problem" sections of the files and creates a global problem P that bounds them from below. It finds a global solution S. The local compilation determines what boundary values of the solution are needed to optimize its translation unit, and write requests for such to the "need" section of the .opa file. The IPA solver reads this information, and writes out the boundary values to the "known" section of the .ipa file.

[0050] Figure 3 is a flow chart illustrating a process 230 to prune local graphs according to one embodiment of the invention.

[0051] Upon START, the process 230 applies a shrinking transform to shrink the local graphs (Block 310). The shrinking transform scans some special cases to perform preliminary graph reduction. The shrinking transform is described in Figure 4A. Next, the process 230 associates a use attribute to each vertex in each of the local graphs (Block 320). The use attribute for a vertex v, when asserted, indicates that there is some named vertex u such that there is an edge u → v. Next, the process 230 associates an affect attribute to each vertex in each of the local graphs (Block 330). The affect attribute for a vertex u, when asserted, indicates that there is some named vertex v such that there is an edge u → v. In one embodiment, the use and affect attributes are Boolean variables where true and false values indicate that the underlying attribute is asserted and

negated, respectively. The use and affect attributes of a vertex u may be denoted as u.uses_named_vertex and u.affects_named_vertex, respectively.

[0052] Then, the process 230 pre-solves a subgraph of each of the local graphs (Block 340). The subgraph includes subgraph edges. Each of the subgraph edges connects a tail vertex to a head vertex where the tail vertex has a negated use attribute. Pre-solving the subgraph is solving a greatest fix-point problem for the subgraph of the local graph, yielding a solution SOL(w), and assigning w.val := SOL(w) for each vertex w that is the head or tail of an edge in the subgraph. Doing so changes the values of the vertices such that the contribution of the local graph problem to the global graph problem is the same even if the subgraph's edges and anonymous vertices are removed. The purpose of the pre-solver is to push the constraints from parts of the graph that will be omitted from the global graph problem, which will be solved by the IPA solver, to parts of the graph that will be included in the global problem.

[0053] Next, the process 230 applies the shrinking transform again to the local graphs (Block 350). This is because the new values for the vertices may enable more opportunities for shrinking. In particular, the presolver may have set more vertices' values to bottom, thus permitting more edges to be removed by the shrinking transform as will be illustrated in Figure 4B.

[0054] Then, the process 230 determine the final edges to be sent to the IPA solver (Block 360). The final edges are those edges of the form u → v where the affect attribute of u and the use attribute of v are asserted. Next, the process 230 determine the final vertex values to be sent to the IPA solver (Block 370). The final vertex values are those values that are different from the lattice top because values of top contribute no information and therefore can be elided.

Since most of the vertex values are top, this saves space in the stored representation of the graph. The final edges and the final vertex values form the pruned local graphs. Next, the process 230 sends the final edges and final vertex values as pruned local graphs to the IPA solver (Block 380). The process 230 is then terminated.

[0055] Figure 4A is a flowchart illustrating the process 310 to shrink the local graphs according to one embodiment of the invention. Note that the process 310 is optional and is used to further improve the reduction of the local graphs.

[0056] Upon START, the process 310 removes each incoming edge having a head value of a lattice bottom (Block 410). Such a value is lattice-bottom in the final solution and therefore the incoming edges add no information. This edge removal is advantageous because it tends to disconnect parts of the graph, which in turn often yields more favorable values (e.g., less "true" values) for the use and affect attributes computed in Blocks 320 and 330 in Figure 2.

[0057] Next, the process 310 transforms each subgraph having edges $u \rightarrow v$, where v is an anonymous vertex, and $v \rightarrow w_i$, to $u \rightarrow w_i$ (Block 420) where $i = 1, \ldots, N$. The edges $u \rightarrow v$ and $v \rightarrow w_i$ have transfer functions $(u \rightarrow v).\ func$ and $(v \rightarrow w_i)$. The values of the vertices u, v, and w_i are u.val, v.val, and w_i.val. Vertex v and both incident edges are removed. For each i, an edge $u \rightarrow w_i$ is added, with transfer function $((v \rightarrow w_i).\ func) \circ ((u \rightarrow v).\ func)$. The value of w_i is set to the lattice meet of $((v \rightarrow w_i).\ func)\ (v.val)$ and the previous w_i.val. If this is not done, the constraint implied by v.val would be lost, possibly resulting in an incorrect answer. This is basically a limited case of the T2 transform as is well known to compiler writers, but applied only to anonymous vertices in the local graph. It

cannot be applied when v is a named vertex since v might have more incident edges added later when the global graph is created by the IPA solver.

[0058] Figure 4B is a diagram illustrating an edge removal in shrinking the local graphs according to one embodiment of the invention. The example shows an edge 430 with a tail vertex 432 and a head vertex 434. The head vertex 434 has a bottom value. The shrinking transform removes the edge 430.

[0059] Figure 4C is a diagram illustrating a subgraph transformation in shrinking the local graphs according to one embodiment of the invention.

[0060] The original subgraph includes a vertex u 442 with value u.val, an anonymous vertex 444 with value v.val, an edge 440 with transfer function e.func connecting vertex u 442 and vertex v 444, vertices w_1, w_2, and w_3 462, 464, and 466 with values w_1.val, w_2.val, and w_3.val respectively, edges 452, 454, and 456 with transfer functions f_1.func, f_2.func, and f_3.func, connecting vertex v 444 to vertices w_1 462, w_2 464, and w_3 466, respectively.

[0061] After the shrinking transform, the vertex v 444 and edges 440, 452, 454, and 456 are removed. Edges 472, 474, and 476 are added to connect vertex u 442 and vertices 482, 484, and 486, respectively. Edges 472, 474, and 476 have transfer functions as the combination of the e.func and f_1.func, f_2.func, and f_3.func, respectively. In other words, the edges 472, 474, and 476 have transfer functions f_1.func o e.func, f_2.func o e.func, and f_3.func o e.func, respectively, where o is function composition. The values of the vertices 482, 484, and 486 are meet (w_1.val, f_1.func (v.val)), meet (w_2.val, f_2.func (v.val)), and meet (w_3.val, f_3.func (v.val)), respectively.

[0062] Figure 5 is a pseudo code illustrating the process 320 to associate a use attribute shown in Figure 3 according to one embodiment of the invention.

[0063] The process 320 essentially determines which vertices' values depend upon other named vertices' values. In other words, for each vertex v, is there some named vertex u such that there is a path from u to v? The process 320 first negates the use attributes for all vertices in the local graph. Then the process 320 invokes a mark use operation on u for each named vertex u in the local graph. In the mark use operation, the process 320 asserts the use attribute associated with u if the use attribute is negated. Then, for each edge connecting the named vertex u to a vertex v, the process 320 recursively invokes the mark use operation on v.

[0064] Figure 6 is a pseudo code illustrating the process 330 to associate an affect attribute shown in Figure 3 according to one embodiment of the invention.

[0065] The process 330 essentially determines which vertices' values affect named vertices' values. In other words, for each vertex u, is there some named vertex v such that there is a path from u to v? The process 330 first negates the use attributes for all vertices in the local graph. Then, the process 330 invokes a mark affect operation on v for each named vertex y in the local graph. In the mark affect operation, the process 330 asserts the use attribute associated with v if the use attribute is negated. Then, the process 330 recursively invokes the mark affect operation on u for each edge connecting the vertex u to a named vertex v.

[0066] Figure 7 is a diagram illustrating an example for the pre-solving shown in Figure 3 according to one embodiment of the invention.

[0067] Note that the subgraph includes all edges u→v such that the u.uses_named_vertex = false; i.e., the vertex u having negated use attribute. In this example, information in the white vertices (those with u.uses_named_vertex = false; i.e., a negated use attribute) is propagated to the gray vertices. This is done by solving the greatest fix-point problem on the subgraph with solid edges, and setting w.val to the pre-solver's solution SOL(w) for each vertex w that is the tail or head of an edge in the subgraph. The reason for solving a subgraph and not the entire graph is that most of the gray vertices usually have a value of lattice-top, and Block 370 in Figure 3 elides these values in the compressed representation of the graph. Solving for the entire graph would cause many of these values to become something other than top, thus requiring that their values be explicitly represented, resulting in uncompression.

[0068] Figure 8 is a diagram illustrating an example of a three-point value lattice according to one embodiment of the invention. The lattice elements are EXTERNAL, STATIC, and UNREFERENCED, with EXTERNAL ≤ STATIC and STATIC ≤ UNREFERENCED.

[0069] Figure 9 is a diagram illustrating an example in the C programming language of two translation units according to one embodiment of the invention.

[0070] Figure 10 is a diagram illustrating the local graphs for the two translation units shown in Figure 9 according to one embodiment of the invention.

[0071] The double circles are the "needed" set. The value of a vertex is "needed" by a translation unit if the corresponding file-scope entity is defined in the translation unit and possibly exported to another translation unit. The global

solution value for said vertex will indicate whether the entity can be removed or given static linkage.

[0072] The vertices for objects "f" and "a" are labeled with anonymous symbols @f and @a, respectively. They are anonymous because the corresponding entities have static linkage and thus cannot be seen directly outside their respective translation units. The values for anonymous vertices are not needed, because their values can be computed from boundary information. The lattice values at the vertices are all UNREFERENCED (top of lattice), except for "main", which a priori is known to be implicitly referenced, and thus gets a value of EXTERNAL.

[0073] Figure 11 is a diagram illustrating the transfer functions of the local graphs shown in Figure 10 according to one embodiment of the invention.

[0074] The lattice points are abbreviated by their initial letter (U, S, and E). For instance, the edge from "d" to "e" (@d → e) maps the lattice value UNREFERENCED to UNREFERENCED, and other lattice values to STATIC. The rationale is that if "f" is referenced in the program, then "e" is indirectly referenced via "f". The other lattice values are mapped to STATIC since the reference is between objects within the same translation unit.

[0075] Figure 12 is a diagram illustrating pruning the local graphs shown in Figure 10 according to one embodiment of the invention.

[0076] The reduced graphs show the result after the shrinking transform. In particular, the subgraph transformation is applied to remove the anonymous vertices @f and @a. The double circles denote named vertices. Because all the

anonymous vertices are removed, all the remaining ones end up being "gray", and the presolver step becomes vacuous.

[0077] Figure 13 is a diagram illustrating the transfer functions of the pruned local graphs shown in Figure 12 according to one embodiment of the invention.

[0078] Figure 14 is a diagram illustrating another example for pruning the local graph for a variant of the second translation unit, according to one embodiment of the invention.

[0079] In this example, suppose b had been an anonymous vertex, i.e., if routine "b" from Figure 9 had been "static", then the vertex "b" would be white instead of gray, because its information flows to (or affects) a named vertex "d", but "b" itself does not use any named vertex. Note that vertices prefixed by @ are unnamed. Therefore, running the pre-solve for the variant of the second translation unit would solve only the subgraph consisting of vertices "b" and "d", and edges b→b and b→d. The fixed-point solver finds that the solution is "val(b) = UNREFERENCED" and "val(d) = UNREFERENCED".

[0080] Per block 370 of Figure 3, only edge c→e would be sent to the IPA solver. The values of c and e would not be sent because they are "top".

[0081] Figure 15 is a pseudo code illustrating the process 340 to presolve a subgraph according to one embodiment of the invention. It integrates the calculation of SOL(w) with setting w.val:=SOL(w). i.e., it does the solution "in place" by operating directly upon w.val for each relevant vertex w; thus SOL(w) is not explicit in the steps. The process 340 has an outer loop and an inner loop. The outer loop iterates until the inner loop causes no changes. The inner loop

iterates over each edge u→v in the graph, and if u has a negated uses_named_vertex attribute, computes the meet of v.val and the edge's transfer function applied to u.val. If the result is not the same as v.val, then v.val is set to the result, and the change noted, so that the outer loop will reiterate. General processes for solving fix-point problems on graphs are well-known to compiler writers. The novelty here is how only some of the edges are considered during pre-solving.

[0082] While this invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications of the illustrative embodiments, as well as other embodiments of the invention, which are apparent to persons skilled in the art to which the invention pertains are deemed to lie within the spirit and scope of the invention.